CQUPT – University at Albany

Computer Science – International College

**ICSI 403 --- Design and Analysis of Algorithms**

**Project 1 --- Spring 2025**

**Table of Contents**

1. **System documentation -------------------------------------------------------------1**
   * A high-level data flow diagram for the system-----------------------------1
   * A list of routines and their brief descriptions-------------------------------2
   * Binary Search Tree-------------------------------------------------------------3
2. **Test documentation------------------------------------------------------------------7**
   * How you tested your program -----------------------------------------------7
   * Testing outputs -----------------------------------------------------------------7
3. **User documentation ----------------------------------------------------------------12**
   * How to run your program ---------------------------------------------------12
   * Describe parameter (if any) -------------------------------------------------13
4. **Source Code--------------------------------------------------------------------------13**

**I.** **System documentation**

**i.** **A high-level data flow diagram for the system**

图示

AI 生成的内容可能不正确。

**ii. A list of routines and their brief descriptions**

|  |  |
| --- | --- |
| **Routine** | **Description** |
| encodeWithParity | Encodes a character with odd parity. |
| encodeInputFile | Encodes a text file into a binary file. |
| decodeWithParity | Decodes an 8-bit binary string with odd parity. |
| decodeResultFile | Decodes a binary file into a text file. |
| Symbols::performOperation | Performs arithmetic operations on a symbol's value. |
| Symbols::toString | Converts a symbol's identifier and value to a string. |
| BinarySearchTree::insert | Inserts a symbol into the BST. |
| BinarySearchTree::find | Finds a symbol in the BST by its identifier. |
| BinarySearchTree::inorder | Performs an in-order traversal of the BST to extract sorted symbols. |
| BinarySearchTree::getAllSymbolsSorted | Returns all symbols in the BST sorted lexicographically. |
| processFile | Processes a binary file and updates the BST. |

**iii. Binary Search Tree**

1. **The BST node structure is defined as follows:**

struct TreeNode

{

    Symbols data;    // Stores the symbol (identifier and value)

    TreeNode \*left;  // Left child node

    TreeNode \*right; // Right child node

    TreeNode(Symbols sym) : data(sym), left(nullptr), right(nullptr) {}

};

* Each node stores a Symbols object, which contains an identifier (identifier) and a value (value).
* The left and right child nodes are used to maintain the BST structure.

**b. BST Class**

The BST class encapsulates operations such as insertion, search, and traversal:

class BinarySearchTree

{

private:

    TreeNode \*root; // Root node of the BST

    // Helper function to insert a node

    TreeNode \*insert(TreeNode \*node, Symbols sym)

    {

        if (node == nullptr)

            return new TreeNode(sym); // If the node is null, create a new node

        // Insert into the left or right subtree based on the identifier

        if (sym.identifier < node->data.identifier)

            node->left = insert(node->left, sym);

        else if (sym.identifier > node->data.identifier)

            node->right = insert(node->right, sym);

        return node; // Return the updated node

    }

    // Helper function to find a node by identifier

    TreeNode \*find(TreeNode \*node, const std::string &id)

    {

        if (node == nullptr || node->data.identifier == id)

            return node; // If node is found or null, return it

        // Search in the left or right subtree based on the identifier

        if (id < node->data.identifier)

            return find(node->left, id);

        else

            return find(node->right, id);

    }

    // Helper function for in-order traversal (to extract sorted symbols)

    void inorder(TreeNode \*node, std::vector<std::string> &symbolsList) const

    {

        if (node != nullptr)

        {

            inorder(node->left, symbolsList);             // Traverse left subtree

            symbolsList.push\_back(node->data.toString()); // Add current node's data to the list

            inorder(node->right, symbolsList);            // Traverse right subtree

        }

    }

public:

    BinarySearchTree() : root(nullptr) {} // Constructor initializes root to null

    // Public function to insert a symbol into the BST

    void insert(Symbols sym)

    {

        root = insert(root, sym); // Call the private insert function

    }

    // Public function to find a symbol by identifier

    Symbols \*find(const std::string &id)

    {

        TreeNode \*node = find(root, id); // Call the private find function

        if (node != nullptr)

            return &node->data; // Return the symbol if found

        return nullptr;         // Return null if not found

    }

    // Public function to get all symbols sorted lexicographically

    std::vector<std::string> getAllSymbolsSorted() const

    {

        std::vector<std::string> symbolsList;

        inorder(root, symbolsList);                        // Perform in-order traversal to extract symbols

        std::sort(symbolsList.begin(), symbolsList.end()); // Sort the symbols lexicographically

        return symbolsList;                                // Return the sorted list

    }

};

1. **How BST Used in the Producer**
2. **Processing Algebraic Expressions**:
   * The processFile function reads the decoded algebraic expressions from the binary file.
   * For each expression, it either: 1 Finds the symbol in the BST using find and updates its value using performOperation. 2 If the symbol does not exist, it creates a new Symbols object, performs the operation, and inserts it into the BST using insert.

void processFile(const std::string &filename, BinarySearchTree &bst)

{

    // Decode the binary file into a temporary text file

    string outputFile = "temp1.txt";

    decodeResultFile(filename, outputFile);

    std::ifstream file(outputFile);

    if (!file.is\_open())

    {

        std::cerr << "Failed to open file: " << filename << std::endl;

        return;

    }

    std::string line;

    while (std::getline(file, line)) // Read each line

    {

        std::istringstream iss(line);

        std::string identifier, operation;

        int value;

        iss >> identifier >> operation >> value; // Parse the line

        Symbols \*sym = bst.find(identifier); // Find the symbol in the BST

        if (sym)

            sym->performOperation(operation, value); // Perform the operation if found

        else

        {

            Symbols newSym(identifier, 0);             // Create a new symbol with initial value 0

            newSym.performOperation(operation, value); // Perform the operation

            bst.insert(newSym);                        // Insert the new symbol into the BST

        }

    }

    file.close();

}

1. **Sorting and Output**:

1.After processing all expressions, the getAllSymbolsSorted function is called to retrieve all symbols in lexicographical order.

2.The sorted symbols are written to a file and encoded into the final binary file (project1Result.outf).

void writeSortedResultsToFile(const vector<string> &sortedSymbols)

{

    string outputFile = "temp2.txt";

    ofstream outFile(outputFile);

    if (!outFile.is\_open())

    {

        cerr << "Failed to open file: " << outputFile << endl;

        return;

    }

    for (const auto &sym : sortedSymbols)

        outFile << sym << endl; // Write sorted symbols to file

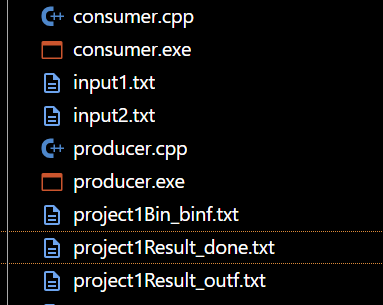
    outFile.close();

    encodeInputFile(outputFile, "project1Result\_outf.txt"); // Encode the results into a binary file

}

**II Test documentation**

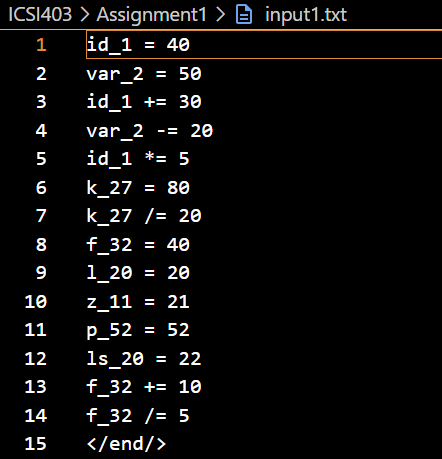
**i.** **How you tested your program**

I execute my program in Vscode, the program structure as follows

I have four test files to test the program, after getting the results, I compare them with the answer which is calculated by myself.

**ii. Testing outputs**

Input1.txt



project1Bin\_binf.txt

背景图案

AI 生成的内容可能不正确。

project1Result\_outf.txt

图片包含 游戏机, 键盘, 电脑

AI 生成的内容可能不正确。

Result

文本

AI 生成的内容可能不正确。

Input2.txt

图片包含 文本

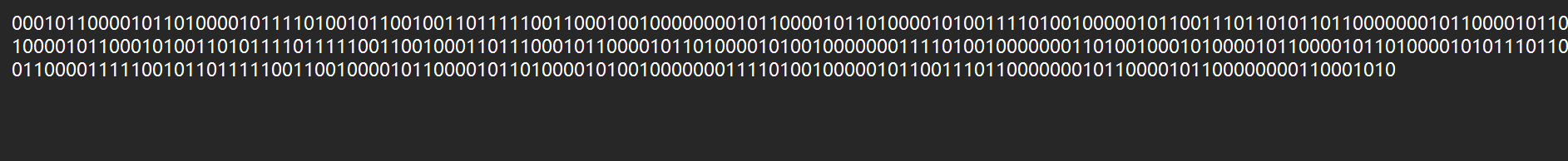
AI 生成的内容可能不正确。

project1Bin\_binf.txt

文本

AI 生成的内容可能不正确。

project1Result\_outf.txt



Result

文本

AI 生成的内容可能不正确。

Input3.txt

电脑萤幕画面

AI 生成的内容可能不正确。

project1Bin\_binf.txt

图片包含 文本

AI 生成的内容可能不正确。

project1Result\_outf.txt

电脑萤幕

AI 生成的内容可能不正确。

Result

文本

AI 生成的内容可能不正确。

Input4.txt

图片包含 文本

AI 生成的内容可能不正确。

project1Bin\_binf.txt

背景图案

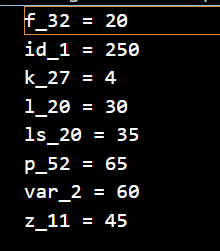
AI 生成的内容可能不正确。

project1Result\_outf.txt

图片包含 游戏机, 电脑, 键盘

AI 生成的内容可能不正确。

Result



**III. User documentation**

**i. How to run your program**

1. Run consumer which achieve input file (input1.txt , input2.txt, input3.txt, input4.txt) and encode them, it will output project1Bin\_binf.txt

文本

AI 生成的内容可能不正确。

1. Run producer (don’t need input) which will receive project1Bin\_binf.txt as input and decode them and do calculation, then it will encode the result and put it into project1Result\_outf.txt
2. Run consumer again, it decode the project1Result\_outf.txt and put the result into project1Result\_done.txt

文本

AI 生成的内容可能不正确。

**ii.** **Describe parameter (if any)**

When running consumer, one can input 0 to choose encode file or 1 to decode file.

**IV. Source Code**

**Correctness:**

I execute my program to test the four example files, and the results are all correct

Layering. Readability. Comments are showing follows

**Layering Readability Comments Efficiency** are showing as follows

**Consumer**

#include <iostream>

#include <fstream>

#include <bitset>

#include <string>

#include <vector>

using namespace std;

const int MAX\_MESSAGE\_SIZE = 5;        // Max characters per block

const string SYN = "0001011000010110"; // Odd parity encoding of ASCII 22 (SYN)

// Encode a character with odd parity

string encodeWithParity(char c)

{

    bitset<7> bits(c);

    int parity = bits.count() % 2 == 0 ? 1 : 0;      // Calculate parity bit

    return bitset<8>((parity << 7) | c).to\_string(); // Combine parity and data

}

// Encode input file and save to binary file

void encodeInputFile(const string &inputFile, const string &outputFile)

{

    ifstream in(inputFile);

    ofstream out(outputFile);

    vector<string> block; // Store encoded characters

    char c;

    while (in.get(c))

    {

        if (c == '<')

            break; // Stop if '<' is encountered

        block.push\_back(encodeWithParity(c)); // Encode and add to block

        if (block.size() == MAX\_MESSAGE\_SIZE) // If block is full

        {

            out << SYN;                            // Write SYN marker

            out << encodeWithParity(block.size()); // Write block size

            for (const auto &encodedChar : block)

                out << encodedChar; // Write characters

            block.clear();          // Clear block

        }

    }

    // Write remaining characters if any

    if (!block.empty())

    {

        out << SYN;

        out << encodeWithParity(block.size());

        for (const auto &encodedChar : block)

            out << encodedChar;

    }

}

// Decode an 8-bit character with odd parity

char decodeWithParity(const string &encodedChar)

{

    string dataBits = encodedChar.substr(1, 7); // Extract 7 data bits

    bitset<7> bits(dataBits);

    return static\_cast<char>(bits.to\_ulong()); // Convert to character

}

// Decode binary file and save to text file

void decodeResultFile(const string &inputFile, const string &outputFile)

{

    ifstream in(inputFile);

    ofstream out(outputFile);

    string content((istreambuf\_iterator<char>(in)), {}); // Read file content

    string buffer;

    // Filter non-binary characters

    for (char c : content)

        if (c == '0' || c == '1')

            buffer += c;

    size\_t pos = 0;

    while ((pos = buffer.find(SYN)) != string::npos) // Find SYN marker

    {

        buffer = buffer.substr(pos + 16); // Skip SYN

        if (buffer.size() < 8)

            break; // Check if enough bits remain

        string lengthEncoded = buffer.substr(0, 8);      // Extract block size

        int blockSize = decodeWithParity(lengthEncoded); // Decode block size

        buffer = buffer.substr(8);                       // Remove block size from buffer

        for (int i = 0; i < blockSize; i++) // Decode each character in block

        {

            if (buffer.size() < 8)

                break;

            string encodedChar = buffer.substr(0, 8); // Extract character

            buffer = buffer.substr(8);                // Remove character from buffer

            char decodedChar = decodeWithParity(encodedChar); // Decode character

            out.put(decodedChar);                             // Write to output file

        }

    }

}

int main()

{

    int choice;

    string inputFile, outputFile;

    cout << "Choose operation (0-encode, 1-decode): ";

    cin >> choice;

    if (choice == 0) // Encode

    {

        cout << "Enter input file name: ";

        cin >> inputFile;

        outputFile = "project1Bin\_binf.txt"; // Output binary file

        encodeInputFile(inputFile, outputFile);

        cout << "Encoding complete. Result saved in " << outputFile << endl;

    }

    else if (choice == 1) // Decode

    {

        inputFile = "project1Result\_outf.txt";  // Input binary file

        outputFile = "project1Result\_done.txt"; // Output text file

        decodeResultFile(inputFile, outputFile);

        cout << "Decoding complete. Result saved in " << outputFile << endl;

    }

    else

    {

        cout << "Invalid choice!" << endl;

    }

    return 0;

}

**Producer**

#include <iostream>

#include <fstream>

#include <string>

#include <sstream>

#include <vector>

#include <algorithm>

#include <bitset>

using namespace std;

const int MAX\_MESSAGE\_SIZE = 5;        // Max characters per block

const string SYN = "0001011000010110"; // Odd parity encoding of ASCII 22 (SYN)

// Encode a character with odd parity

string encodeWithParity(char c)

{

    bitset<7> bits(c);

    int parity = bits.count() % 2 == 0 ? 1 : 0;      // Calculate parity bit

    return bitset<8>((parity << 7) | c).to\_string(); // Combine parity and data

}

// Encode input file and save to binary file

void encodeInputFile(const string &inputFile, const string &outputFile)

{

    ifstream in(inputFile);

    ofstream out(outputFile);

    vector<string> block; // Store encoded characters

    char c;

    while (in.get(c))

    {

        if (c == '<')

            break; // Stop if '<' is encountered

        block.push\_back(encodeWithParity(c)); // Encode and add to block

        if (block.size() == MAX\_MESSAGE\_SIZE) // If block is full

        {

            out << SYN;                            // Write SYN marker

            out << encodeWithParity(block.size()); // Write block size

            for (const auto &encodedChar : block)

                out << encodedChar; // Write characters

            block.clear();          // Clear block

        }

    }

    // Write remaining characters if any

    if (!block.empty())

    {

        out << SYN;

        out << encodeWithParity(block.size());

        for (const auto &encodedChar : block)

            out << encodedChar;

    }

}

// Decode an 8-bit character with odd parity

char decodeWithParity(const string &encodedChar)

{

    string dataBits = encodedChar.substr(1, 7); // Extract 7 data bits

    bitset<7> bits(dataBits);

    return static\_cast<char>(bits.to\_ulong()); // Convert to character

}

// Decode binary file and save to text file

void decodeResultFile(const string &inputFile, const string &outputFile)

{

    ifstream in(inputFile);

    ofstream out(outputFile);

    string content((istreambuf\_iterator<char>(in)), {}); // Read file content

    string buffer;

    // Filter non-binary characters

    for (char c : content)

        if (c == '0' || c == '1')

            buffer += c;

    size\_t pos = 0;

    while ((pos = buffer.find(SYN)) != string::npos) // Find SYN marker

    {

        buffer = buffer.substr(pos + 16); // Skip SYN

        if (buffer.size() < 8)

            break; // Check if enough bits remain

        string lengthEncoded = buffer.substr(0, 8);      // Extract block size

        int blockSize = decodeWithParity(lengthEncoded); // Decode block size

        buffer = buffer.substr(8);                       // Remove block size from buffer

        for (int i = 0; i < blockSize; i++) // Decode each character in block

        {

            if (buffer.size() < 8)

                break;

            string encodedChar = buffer.substr(0, 8); // Extract character

            buffer = buffer.substr(8);                // Remove character from buffer

            char decodedChar = decodeWithParity(encodedChar); // Decode character

            out.put(decodedChar);                             // Write to output file

        }

    }

}

// Class to represent symbols (identifiers and their values)

class Symbols

{

public:

    std::string identifier;

    int value;

    // Constructor

    Symbols(std::string id, int val) : identifier(id), value(val) {}

    // Perform arithmetic operations

    void performOperation(std::string operation, int val)

    {

        if (operation == "+=")

            value += val;

        else if (operation == "-=")

            value -= val;

        else if (operation == "\*=")

            value \*= val;

        else if (operation == "/=")

            value /= val;

        else if (operation == "=")

            value = val; // Direct assignment

        else

            std::cerr << "Unknown operation: " << operation << std::endl;

    }

    // Convert symbol to string

    std::string toString() const

    {

        return identifier + " = " + std::to\_string(value);

    }

};

// Binary Search Tree node

struct TreeNode

{

    Symbols data;

    TreeNode \*left;

    TreeNode \*right;

    TreeNode(Symbols sym) : data(sym), left(nullptr), right(nullptr) {}

};

// Binary Search Tree class

class BinarySearchTree

{

private:

    TreeNode \*root;

    // Insert a node into the tree

    TreeNode \*insert(TreeNode \*node, Symbols sym)

    {

        if (node == nullptr)

            return new TreeNode(sym);

        if (sym.identifier < node->data.identifier)

            node->left = insert(node->left, sym);

        else if (sym.identifier > node->data.identifier)

            node->right = insert(node->right, sym);

        return node;

    }

    // Find a node by identifier

    TreeNode \*find(TreeNode \*node, const std::string &id)

    {

        if (node == nullptr || node->data.identifier == id)

            return node;

        if (id < node->data.identifier)

            return find(node->left, id);

        else

            return find(node->right, id);

    }

    // In-order traversal to extract sorted symbols

    void inorder(TreeNode \*node, std::vector<std::string> &symbolsList) const

    {

        if (node != nullptr)

        {

            inorder(node->left, symbolsList);

            symbolsList.push\_back(node->data.toString());

            inorder(node->right, symbolsList);

        }

    }

public:

    BinarySearchTree() : root(nullptr) {}

    // Insert a symbol into the tree

    void insert(Symbols sym)

    {

        root = insert(root, sym);

    }

    // Find a symbol by identifier

    Symbols \*find(const std::string &id)

    {

        TreeNode \*node = find(root, id);

        if (node != nullptr)

            return &node->data;

        return nullptr;

    }

    // Get all symbols sorted by identifier

    std::vector<std::string> getAllSymbolsSorted() const

    {

        std::vector<std::string> symbolsList;

        inorder(root, symbolsList);                        // Extract symbols

        std::sort(symbolsList.begin(), symbolsList.end()); // Sort lexicographically

        return symbolsList;

    }

};

// Process input file and perform operations

void processFile(const std::string &filename, BinarySearchTree &bst)

{

    string outputFile = "temp1.txt";

    decodeResultFile(filename, outputFile); // Decode binary file

    std::ifstream file(outputFile);

    if (!file.is\_open())

    {

        std::cerr << "Failed to open file: " << filename << std::endl;

        return;

    }

    std::string line;

    while (std::getline(file, line)) // Read each line

    {

        std::istringstream iss(line);

        std::string identifier, operation;

        int value;

        iss >> identifier >> operation >> value; // Parse line

        Symbols \*sym = bst.find(identifier); // Find symbol

        if (sym)

            sym->performOperation(operation, value); // Perform operation

        else

        {

            Symbols newSym(identifier, 0); // Create new symbol

            newSym.performOperation(operation, value);

            bst.insert(newSym); // Insert into tree

        }

    }

    file.close();

}

// Write sorted results to file

void writeSortedResultsToFile(const vector<string> &sortedSymbols)

{

    string outputFile = "temp2.txt";

    ofstream outFile(outputFile);

    if (!outFile.is\_open())

    {

        cerr << "Failed to open file: " << outputFile << endl;

        return;

    }

    for (const auto &sym : sortedSymbols)

        outFile << sym << endl; // Write sorted symbols

    outFile.close();

    encodeInputFile(outputFile, "project1Result\_outf.txt"); // Encode results

}

int main()

{

    BinarySearchTree bst;

    processFile("project1Bin\_binf.txt", bst); // Process input file

    std::vector<std::string> sortedSymbols = bst.getAllSymbolsSorted(); // Get sorted symbols

    writeSortedResultsToFile(sortedSymbols);                            // Write results to file

    return 0;

}